AFRL-IF-RS-TR-2004-138
Final Technical Report
May 2004

SCALABLE REDUNDANCY FOR INFRASTRUCTURE SERVICES

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. N402

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-138 has been reviewed and is approved for publication

APPROVED: /s/
KEVIN A. KWIAT
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 2004	**FINAL** **Dec 02 – Dec 03**

4. TITLE AND SUBTITLE

SCALABLE REDUNDANCY FOR INFRASTRUCTURE SERVICES

5. FUNDING NUMBERS

G - F30602-02-1-0139
PE - 63760E
PR - N402
TA - 0A
WU - S1

6. AUTHOR(S)

Michael Reiter

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie Mellon University
Department of Electrical & Computer Engineering
Pittsburgh PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRLIF-RS-TR-2004-138

11. SUPPLEMENTARY NOTES

DARPA Program Manager: Lee Badger/IPTO/(571) 218-4327
AFRL Project Engineer: Kevin A. Kwiat/IFGA/(315) 330-1692 Kevin.Kwiat@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Redundancy is a core tenet of robust system design, and has been employed as a defense against system failure and/or compromise for decades. One goal of this effort is to explore directions toward achieving redundancy that scales to large systems. Another goal of this research is to explore the application of scalable redundancy technologies to the implementation of infrastructure services of a variety of types.

14. SUBJECT TERMS

redundancy, scalability, Byzantine fault tolerance, information assurance

15. NUMBER OF PAGES 39

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

1 Introduction

This 18-month project was a "seedling" effort to generate new research directions in scalable redundancy. Redundancy is a core tenet of robust system design, and has been employed as a defense against system failure and/or compromise for decades. A common example of redundancy as employed to mask the failure of a server, for example, is to employ multiple, synchronized replicas of the server. If a benign failure is to be tolerated, then a primary may typically serve requests, and clients may "fail over" to a backup if the primary fails. To tolerate Byzantine [29] server failures, a strategy is to employ multiple replicas of the server, each initialized identically, and have clients issue queries to all of them, typically in the same order (so that servers retain synchronized state). The client, then, can use the responses from the server replicas as "votes" as to the correct response. This approach is often called "state machine replication", "active replication", or "*n*-modular redundancy" [48].

Systems that employ redundancy and distribution often do not scale well as the number of replicas grows. This can result from many different phenomena. For example, if the number of replicas grows dynamically and opportunistically, then this presents challenging search problems, namely finding suitable hosts for deployment of replicas and, once deployed, for enabling clients to find replicas. As another example, the core protocols underlying a technology like state machine replication tend not to scale well in particular technical measures, such as the message or communication complexity of the query and response protocols. Indeed, in this approach, it is typically the case that adding new servers only slows down the service, since it increases the protocols' obligations e.g., to order requests among more servers.

As a "seedling" effort, our goal was to examine a variety of directions for gaining scalability in systems that employ redundancy. Of the approaches we explored, the two detailed in this report are the directions we matured the most during the duration of this project. The first direction is targeted at the resource location issue above, i.e., to find hosts suitable for the deployment of replicas or to find replicas of servers or objects themselves. So as to make our approach to this problem as general as possible, we abstracted this problem into simply that of keyword search in a network, i.e., to identify hosts holding files containing selected keywords. As these files can include, e.g., object names, service names, or available resources, this technology can easily be specialized to support the location of replicas or suitable hosts for them.

The second effort that we describe here targets the scaling limitations of redundancy protocols themselves. Specifically, we developed new protocols for implementing a Byzantine fault-tolerant data storage service. Our new protocols draw techniques from our prior work on adapting *quorum systems* to mask arbitrary replica failures (corruptions) [31][33]. In this approach, clients interact with only a subset (quorum) of server replicas in each request. In contrast to

state machine replication, this approach can achieve load balancing in a replicated service, and so can scale better. Moreover, we believe our protocols are the first to achieve linearizable data semantics [21] for read-write storage in a scalable fashion.

Both of our efforts have yielded encouraging follow-on developments, one a successful transition to the government and one to a submission to the Self-Regenerative Systems program from DARPA IPTO. First, the tool we developed for distributed keyword search, called Mingle, was adopted by the security officer in National Climate Data Center (NCDC; http:/www.ncdc.noaa.gov) to search distributed network logs for network monitoring and management. NCDC is the world's largest archive of weather data and also does analysis of weather data and current weather trends. Being a government organization, and having multiple high-speed connections and much storage space, they get their fair share of attackers trying to break into their servers. For network monitoring and management, NCDC collects gigabytes of traffic logs (tcpdump files, email server logs, Web server logs, etc.) on numerous distributed computers on a daily basis. The security officer uses Mingle to index and search through these logs quickly to see if an IP address has been to their network before, to look for access patterns for a specific IP address, to search for keywords that correlate individual user's network activities, etc. Since logs are distributed around the center, security is important to prevent unauthorized access. In addition, the speed that Mingle provides and its scalability to search a large intranet also make Mingle appealing for this usage. Though this is not specifically the use of Mingle that we had anticipated, this example is a compelling dual use for this technology, and we are excited that it has found use in the government.

The second follow-on development to the work described here is that the second effort described above has formed a starting point for a new submission to the Self-Regenerative Systems program from DARPA IPTO, which we have been given preliminary indication will be funded. To the extent that such indications of government interest are the measure of success for a “seedling” effort such as this, we believe this project has been fruitful.

1.1 Publications

Papers describing portions of the work in this project were or will be presented at the 11th IEEE International Symposium on High Performance Distributed Computing and the 2004 International Conference on Dependable Systems and Networks.

1.2 Patents

There were no patents filed as a result of this project.

1.3 Financial Summary

The full budget of $588,539.00 was expended during the performance period. This money was used primarily to support graduate students working on the project, as well as faculty members involved in this project.

2 A Secure Distributed Search System

Although there are a number of useful tools for locating data on a single machine, locating data in a distributed environment can be troublesome. Tools like `grep` and `find` are good for searching small directory hierarchies, but are inappropriate for searching entire disks. The GNU `locate` command provides fast keyword search of file names, but not the contents of those files. Tools, such as Glimpse [34] and Windows Indexing Service, precompute inverted index tables of local files. Each entry in the index table stores a word and its occurrences in the files, which enables fast keyword querying. However, these tools do not support querying across different computers. Peer-to-peer applications such as Napster[1], Gnutella[2] and Freenet [10] have been used for large-scale locating and sharing of MP3 files. A drawback of such systems is that there is no security mechanism to protect data from access by unauthorized users. In addition, no indexing is provided to quickly locate information.

Thus, we believe we need new systems that help people and applications find data in distributed computing environments. These systems should be both efficient, in the sense that searches complete quickly, and secure, in the sense that unauthorized users are not allowed to locate data. So how might we build such systems?

A straightforward solution is to build a global indexing service, where dedicated servers crawl files from every computer on the Internet and then compute a centralized index table. Search engines like Google[3] have used this kind of scheme very effectively for the Web. However, the centralized model is inappropriate for searching personal computing systems and other nonpublic data for a number of reasons. First, indexing is an expensive operation requiring large amounts of memory and disk space. Even massive search engines like Google can index only limited number of Web pages. Therefore, centralized indexing servers do not scale with the increasing number of computers and the exploding capacities of modern disks. Second, many personal files are private in nature. Users lose control of their files once they are indexed by the server. Even with complicated security and access control mechanisms, they may be unwilling to release their files to remotely administered servers.

Another approach is to have one or more dedicated indexing servers for a cluster of computers. For example, distributed search engines such as Harvest [4] set up one or more index servers to search within an intranet. With this scheme, a significant amount of network traffic is needed to fetch distributed files to the servers for index computing. More important, this scheme requires large,

[1] http://www.napster.com
[2] http://www.gnutella.com
[3] http://www.google.com

expensive, dedicated indexing servers, which are not feasible in a personal computing environment.

In this section, we present Mingle, a secure distributed search system. Mingle is designed to meet the following requirements, which we consider fundamental to a distributed search system for personal computing and sensitive applications:

- **Searches should be fast**. For fast search, Mingle precomputes an inverted index of local files on each participating host. A query can be processed by the local host, or routed through the participating hosts using peer-to-peer communication to locate all of the desired data.

- **The system should scale**. Since each participating host devotes computing resources for indexing, the Mingle scheme should scale well with the number of computers. Participating hosts communicate with each other only when there is a search request, greatly reducing network traffic.

- **The system should be secure**. The Mingle security architecture focuses on preventing unauthorized release of information while allowing files to be maximally shared. Since search is presumably a frequent operation, we insist that the security mechanism be as convenient as possible for users. Access control policy is expressed using an *access-right mapping*, a novel mechanism that extends a local file system's access control primitives to Mingle users in a uniform and convenient way. This mechanism builds upon a single sign-on mechanism implemented in Mingle, which allows a user to perform authenticated search requests across many Mingle servers seamlessly.

The remainder of this section is organized as follows. Section 2.1 summarizes related work. Sections 2.2, 2.3, and 2.4 describe the Mingle prototype, including the novel security architecture based on access-right mapping. A preliminary performance evaluation of the Mingle prototype is in Section 2.5.

2.1 Related Work

Distributed search has been studied in the area of information retrieval [6][15][25][54], with emphasis on algorithms for server selection and result merging. Mingle is different from these works in that our focus is on the system architecture and the security mechanisms to prevent data from access by unauthorized users.

Peer-to-peer systems [44][51][47][58] have been designed to locate objects in self-organizing overlay networks. Such systems use hash based distributed indexing schemes to locate objects. The location of each object is stored at one or more nodes selected by a distributed hash function. Although hash functions can deterministically locate an object, they do not support keyword searching.

2.2 Overview of the Mingle System

On each host, there is a Mingle server running as a daemon. Communication among servers is peer-to-peer. A user may issue a request from any host to any of the servers by launching a lightweight client program, which simply sends the request to the local server and waits for replies. If only a local reply is required, then the local server handles the request and sends back the reply. Otherwise, the server forwards the request to remote servers for further processing.

Mingle clients issue separate requests for indexing and searching. Only the owners of files can issue requests to index those files. Any Mingle user can issue a search request to any Mingle server, but they only receive information about files that they are authorized to see, as described in the next section.

When a Mingle server daemon is started on a host for the first time, none of the files on that host are indexed. Users must make explicit requests to the Mingle server to index directory trees that they own. Thus Mingle is “opt-in”, in the sense that users on Mingle hosts must issue explicit requests before their data is indexed and made available to Mingle clients.

Each Mingle server computes an inverted index of local files that have been indexed. The inverted index consists of: (1) a lexicon containing all of the words that appear in the files; and (2) an inverted file entry for each word, which stores a list of pointers to the occurrences of that word in the files. To locate a given word, only its inverted file entry needs to be traversed, allowing fast queries. The detailed design and data structures of the inverted index in Mingle are discussed in Section 2.4.1.

In many cases, a user may wish to search all hosts in a Mingle cluster without specifying host identities. To enable this, we establish a *master server*, which is a normal Mingle server that maintains the list of host names inside the cluster. Upon reception of a user request that needs to be routed through the cluster, the local server first fetches the host list from the master server, and then forwards the request to each remote host in the list individually.

While the server is implemented as a daemon, the client program is a lightweight program that is launched only when needed. An alternative design is for each user to run their own stand-alone server that they interact with via the command line. This approach would lead to multiple server processes running simultaneously on the same computer, leading to a large overhead. By implementing a separate lightweight client program, we build one index table and enable multiple users on the same host to share a single Mingle server.

2.3 Mingle Security Architecture

The Mingle security architecture focuses on preventing unauthorized release of information while allowing files to be shared among different types of users. Both the *local users* who have accounts on the Mingle host and *remote users* who do not have accounts should be able to participate in Mingle. Sensitive files can be accessed only by authorized users, while public files can be searched by even anonymous users.

Existing mechanisms are deficient in terms of both flexibility and user convenience. Mingle users might belong to different organizations and not have accounts on every machine. In this case, file system access controls on the Mingle hosts are not flexible enough to separate remote visitors into different classes of trustworthiness. Further, search is a stateless request involving one simple command. Supplying passwords with each request is not acceptable.

The design of the Mingle security architecture is guided by the following three principles:

- **File owners decide whom to trust**. We cannot expect every Mingle user to trust the same set of people. We must let file owners decide who is allowed to access their files.

- **Authorization is flexible and convenient**. Because some files are more sensitive than others and some people are more trustworthy than others, file owners must be able to specify access rights for different users on each single file conveniently.

- **Authentication has small overhead**. Since "search" is a stateless operation for everyday usage, we require the user authentication mechanism to be as lightweight as possible while providing reasonable level of security.

In the following, we present the details of the Mingle security architecture. We begin by describing the authorization mechanism, which addresses the first and second principles. We then discuss the user authentication mechanism, which addresses the third design principle. We close this subsection with a discussion of possible malicious attacks against Mingle.

2.3.1 Authorization

A straightforward way to handle access control is to maintain an access control list (ACL) for each file. Each item in the ACL specifies the permitted operations for each user. Although ACLs are flexible, they can be costly and prone to error since file owners must manually specify an ACL for each file.

We propose a new, more convenient approach that arises from the observation that the underlying file system in the Mingle host already enables access control on each file. By granting a user (or group) "read" permission to a file, the file owner implicitly allows that user (or group) to search the file as well. However, the access control schemes in the file system are only applicable to local users who have accounts on the same computer.

To extend the file system access control to a remote user, we introduce the idea of *access-right mapping*. For a file owner with local account name *A*, a Mingle user with Mingle ID *U* (see Section 2.3.2) can be mapped to a *search protection domain* (SPD) that consists of one or more local users or user groups:

$SPD_A(U)$ = {userID1, userID2, …, groupID1, groupID2, …},

where userID*i* is some local user account name, and groupID*i* is some local group name.

The meaning of the mapping is that Mingle user *U* has permission to search any local file owned by *A* and readable by one or more members of $SPD_A(U)$.

The process of access-right mapping is performed by each file owner independently. Thus, a Mingle user can be mapped to different SPDs by different file owners on the same host. Given the access-right mapping, the algorithm for access permission checking is simple, as shown below.

```
// Return whether Mingle user U is allowed to search file F
bool is_search_permitted(filename F, mingle_user U)  {

    // Get the file owner of F
    O = get_file_owner(F);

    // Get the SPD of U with respect to the file owner O
    SPD = O.get_SPD(U);

    // Check if any member in SPD is allowed to read F
    foreach id in SPD {
        if F is readable by id {
            return true;
        }
    }

    return false;
}
```

The access-right mapping preserves the file system access control semantics. It greatly simplifies the access control specification, while giving file owners full

control over their data. For example, each Unix file has 9 mode bits associated with it. These mode bits specify whether the file owner, specific group of users, and everyone else can read, write or execute the file. In many cases, a file owner can map a friendly remote Mingle user to an SPD that consists of only the owner account or a "guest" account, allowing file owners to specify access permissions to most of their files conveniently. For the small number of files that need fine-grained access control, file owners can define a user group for each file and map Mingle users to the corresponding user groups. In particular, in order to allow anonymous Mingle users to search shared public files, a file owner can define a special user group for public files and map any anonymous user to that group.

2.3.2 User Authentication

In Mingle, each request to index or search files on a host must be authenticated by that host. Since a Mingle user might not have accounts on every host, or might have different account names on different hosts, each Mingle user is assigned a unique global Mingle ID (a text string) that identifies the user to Mingle servers in a uniform way. A Mingle user without a Mingle ID is regarded as an anonymous Mingle user and can search only public files.

A Mingle ID is assigned to a user via a registration process that she executes once. In this registration process, the user selects and inputs a Mingle ID and password to her lightweight client, which conveys these inputs to the local Mingle server. The local Mingle server sends this pair to the master server for this Mingle cluster, using an encrypted channel (e.g., encrypted under the public key of the master server). The master server confirms that this Mingle ID has not previously been registered. If so, it generates a public signing key pair (e.g., [46]) for this Mingle ID, and saves the Mingle ID and associated password and key pair. Upon successful return, the user can convey her Mingle ID to other users in whatever way she wishes, so that these users can create access-right mappings (see Section 2.3.1) for this Mingle ID on other machines, as they choose. An alternative is to use the public key itself, or its hash, as the user's Mingle ID, and then to rely on an external certification infrastructure to convey the identity of the owner of this Mingle ID to others.

This user can then execute distributed searches using Mingle from any computer running a Mingle server as follows. The user enters her Mingle ID, password, and search keyword into the Mingle client, which conveys these to the local Mingle server. The local Mingle server executes a protocol with the master server to retrieve the private key corresponding to this Mingle ID (using the password to authenticate to the master server). Once the private key is obtained, the local Mingle server can issue the query, containing the user's Mingle ID and signed using the retrieved private key, to the relevant remote Mingle servers. Either a certificate binding the Mingle ID to the public verification key can be sent along with this request, or else the remote Mingle server can use

the contained Mingle ID to retrieve the corresponding public key from the master server. Once it has the appropriate public key, it can verify the signature.

There are numerous opportunities to use caching to eliminate steps in the above description and thereby improve the user experience. Specifically, the user's local Mingle server can temporarily cache the user's private key for use in subsequent searches, which eliminates the need for the user to re-enter her Mingle ID or password. Moreover, a remote Mingle server can temporarily cache the public key of this Mingle ID, so that it need not contact the master server again upon receiving another search query bearing this Mingle ID. Of course, this caching also introduces windows of vulnerability: e.g., if the user's public key is revoked due to the compromise of the corresponding private key, this may go unnoticed by a remote Mingle server that is caching the public key. It is therefore necessary to tune this caching to best balance performance, user experience, and security. Such tradeoffs are common in public key infrastructures (e.g., [30]).

A benefit of this architecture is the fact that the user's password and private key are exposed only on machines where the user enters her password (and on the master server). Moreover, the protocol by which the user's machine retrieves the user's private key can be constructed to achieve strong security properties (e.g., see [40]), notably that the protocol messages themselves do not leak information that would permit an eavesdropping adversary to conduct a “dictionary attack” against the user's password [37][26]. As a result, dictionary attacks are limited to online guesses sent to the master server, which the server can detect and stop. The primary vulnerability of this approach is the master server itself: if penetrated, the master server will leak all users’ private keys. This risk can be mitigated by distributing the master server in a way that requires multiple master servers to be compromised to disclose sensitive data (e.g., [14]), though we have not implemented this approach in the present system.

We view the above approach to user authentication and single sign-on in Mingle as an interim solution suitable for Mingle deployments in user populations lacking a unified authentication infrastructure. For user populations with an existing authentication and single sign-on solution, ideally Mingle would exploit that solution for its user authentication needs.

2.3.3 Other Vulnerabilities

We briefly outline types of malicious attacks that Mingle is vulnerable to and discuss possible ways to cope with them. Completely addressing these attacks is beyond the scope of this seedling project.

Mingle query responses are sent from remote servers unencrypted, and thus Mingle is vulnerable to information release and modification attacks. Moreover, without strong authentication of servers, a malicious Mingle server can provide

fraudulent information. If data privacy and integrity is a major concern, then further cryptographic protocols can be used to authenticate servers as well as clients, and to set up session keys for message encryption.

A potential vulnerability to timing attacks exists within Mingle, due to its precomputation of an inverted index to permit fast searching. Specifically, the processing time for a Mingle server to compute its response is a function of the number of files actually containing the search item, not only those to which the client has search access. As a result, a client that can accurately measure the duration required for a Mingle daemon to respond to its search request can learn some information about the number of files on that host that contain the search item, even if the client has search access to very few of them. Randomizing search latencies could mitigate this threat. In addition, a filter could be applied to check user permission before searching through the inverted index. We note, however, that this threat applies only to files that their owners have volunteered to be indexed by Mingle.

Finally, like most other distributed systems, Mingle is vulnerable to various forms of denial-of-service attacks.

2.4 Mingle Implementation

In this section, we discuss the implementation of the Mingle server and client. We first describe the design of the inverted index in Mingle. Then we present the Mingle server architecture and explain the interactions among various system components.

2.4.1 Inverted Index

Indexing is a mechanism for quickly locating a given word in a collection of files. There are three common data structures for file indexing: *inverted index*, *signature files* and *bitmaps* (see [56]). An inverted index is the most natural indexing method, with each entry consisting of a word and its occurrences in the files. A signature file is a probabilistic method for file indexing, where each file has a signature. Every indexed word in a file is used to generate several hash values. The bits of the signature corresponding to those hash values are set to one, indicating the occurrences of the word. A bitmap stores a bit vector for every word. Each bit in the bit vector corresponds to a file and is set to one if the word appears in that file. Compared with the inverted index, signature files can cause false matches, resulting in either longer search times or large signature files. Bitmaps have relatively short search times, but require extravagant storage space and the update is slow when files are updated frequently. In Mingle, we decided to choose the inverted index because of its relatively small cost of storage and low search latency.

However, a fine-grained inverted index is still space consuming. A fine-grained inverted index containing all occurrences of every word can consume 50% to 300% of the original text size, which may not be acceptable. Therefore, Mingle computes a coarse-grained inverted index. Each index entry for a word contains only the first occurrence of that word in every file. A hash table is used to quickly locate the index entry for a word.

Before a file is indexed, it is assigned a document ID. Then the file is scanned word by word to build the index table incrementally. All of the words are converted to lower case. User specified stop words (defined by a configuration file) are removed to reduce index size. For each word in the index table, if it appears multiple times in the file, then only the position of the first occurrence of that word will be recorded in the corresponding index entry. Below is an example of inverted index table in Mingle. Once the index table is built, it can be updated regularly to remove out of date entries.

Word ID	Word	(Document ID; First occurrence)
1	movie	(1;6), (4; 228)
2	day	(2;8), (3;57), (4;200)
3	event	(1;37), (3;22)

A query can consist of one or more keywords. With a coarse-grained inverted index table, queries are resolved in two steps. First, the corresponding index entries of the queried keywords are searched to return a list of files that match the query. Then, each individual file in the list is scanned to return all the exact occurrences of the queried keywords. There is a tradeoff between index granularity and search latency. Compared with the fine-grained inverted index, a coarse-grained index table requires longer search latency since the second step will be otherwise unnecessary. However, the extra latency is typically small, as is shown in Section 2.5.

2.4.2 Mingle Server Architecture

The Mingle server is implemented as a single process. The *file descriptor manager* uses the `select` function to multiplex concurrent requests. After a request has been received by the receiver, it is parsed by the *request manager*, which determines the request type and forwards the request to the appropriate components for further processing. The major components that process a user request are the *file indexer*, *query processor*, and *security manager*. The file indexer accesses files from the local disk and builds up an inverted index table in disk. For performance optimization, the file indexer maintains a cache in memory for frequently accessed terms and their indices. The query processor processes user queries, including advanced query options based on the index table built by the file indexer. The security manager performs access control.

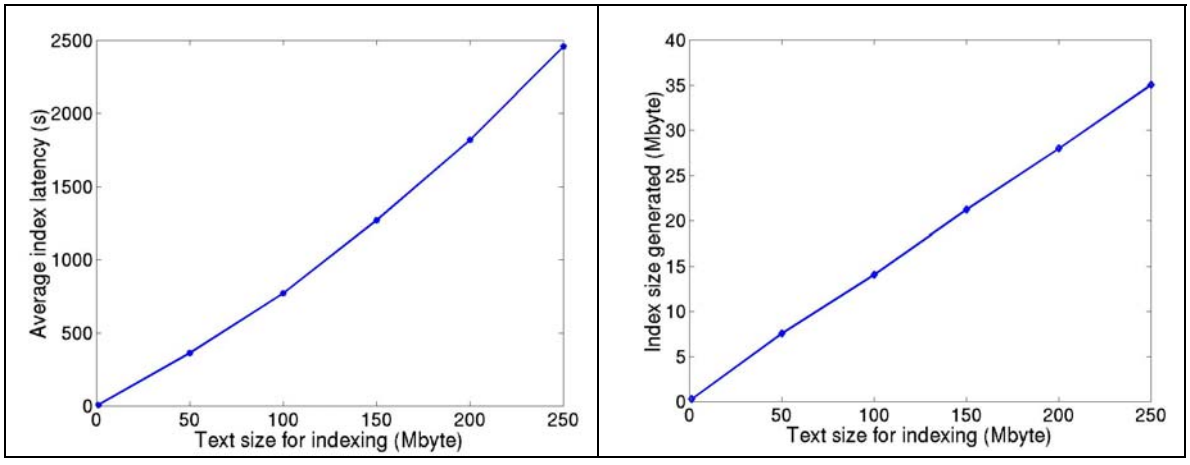
Both the Mingle server and the client program are implemented in C++. The request signing and signature verification use the RSA algorithm [46], which is implemented by the Crypto++ library (version 4.2)[4]. The communications among servers are via TCP connections, while the server and the client program communicate via Unix IPC.

2.5 Performance Evaluation

In this section, we present the performance evaluation of Mingle. We have conducted three sets of experiments to answer the following three questions: (1) What is the cost of index and search—the two major operations in Mingle? (2) What is the impact of our security mechanism on performance? (3) What is the scalability of Mingle? The first and the second sets of experiments are conducted on PIII 550MHz machines with 128 MB of RAM. The last set of experiments is run on the cluster of computers (PIII 550MHz) in a 10BaseT Ethernet LAN. Each data point in the figures is the average of ten runs.

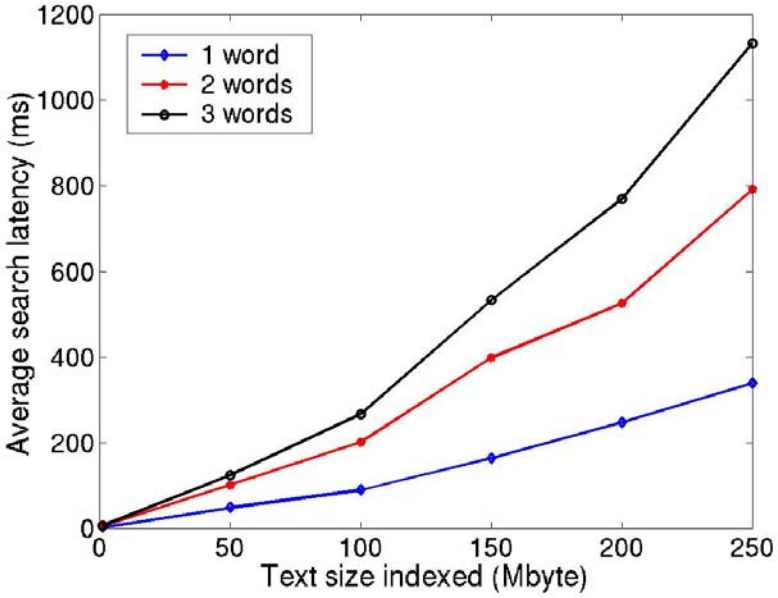
2.5.1 What is the Cost of Index and Search?

Since only text files will be indexed, we have downloaded the RFC[5] and the Internet Drafts[6] repositories to test the index performance. We vary the text size to be indexed. The figures below plot the index latency and the generated index table size. We observe that both costs increase linearly with the text size. It takes about 30 minutes to index 200 MB of text (about 9 seconds per 1 MB). Usually, only a portion of the data on a disk will be text. With the current index speed, we can index a local disk regularly during machine idle time. The generated index table size is about 15% of the original text size.



[4] http://www.eskimo.com/~weidai/cryptlib.html
[5] http://www.rfc-editor.org
[6] http://www.ietf.org/ID.html

With the pre-computed index table, we then examine the search latency on the local server without using our security mechanisms. We vary the indexed text size and the number of keywords in a query. The figure below plots the query lookup latency in the case of cache hits, when the required items in the index table are already in memory. Overall, the search latency is on the order of milliseconds and seconds, which is fast. For example, in 200 MB text, it takes about 250 ms to find answers to a two-keyword query, while it takes as long as 15 seconds to get the same results using `grep`. If the keywords in a query do not exist in the indexed text, the search latency is less than 1ms regardless of the indexed text size.



2.5.2 What is the Impact of Security on Performance?

In this section, we measure the impact of the Mingle security mechanism on performance. Since cryptographic computation is often expensive, our main concern is the latency penalty of cryptographic operations for remote user authentication. We evaluate the cost of request signing and signature verification by measuring the time spent in each step of request processing.

We conducted our experiments on two machines serving as the local server and the remote server respectively in the same LAN. Since the security penalty does not depend on request type, we choose a 3-keyword search request as our example and fix the indexed text size to be 100 MB. We use 1024-bit RSA keys. The table below includes the processing steps we are interested in. The processing consists of two stages: First, the request is parsed and signed at the local server, and forwarded to the remote server. Second, the remote server

verifies the signature and generates the reply by query lookup. The "Total" column corresponds to the time elapsed between the arrival of the request and sending the reply to the client program by the local server. The "Networking" column corresponds to the latency spent in forwarding the request and getting the reply from the remote server. For each step, we show the mean and the standard deviation of latency in milliseconds, as well as the percentage of total latency.

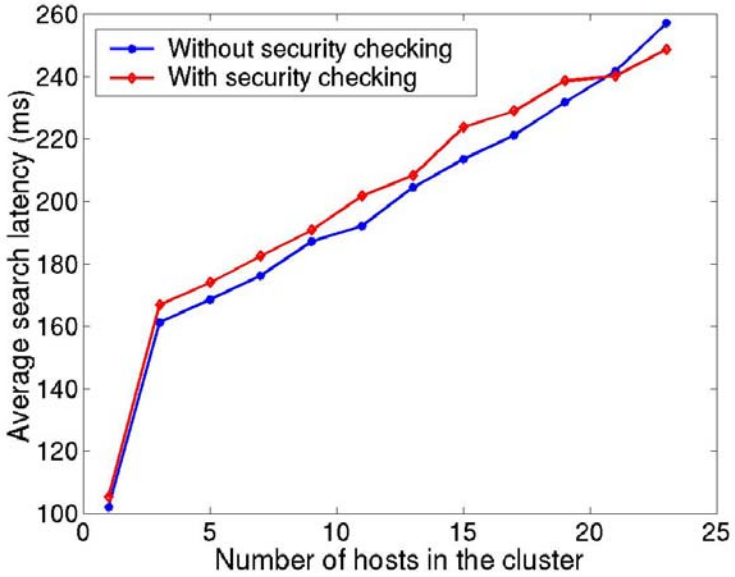
	Total	Parsing	Networking	Lookup	Signing	Sig Verify
Mean	313590	940	6010	279530	25710	1400
Std dev	2615	15	643	2613	44	25
Percentage	100.0%	0.3%	1.9%	89.1%	8.2%	0.5%

We can see from the above table that most of the processing latency is spent on query lookup. Although request signing is also expensive, it is not the performance bottleneck. Compared with signing, signature verification is fast. Note that the standard deviation is small for all steps except networking latency, which has a relatively larger variation due to the network instability. In summary, our security mechanism has little impact on overall search performance.

2.5.3 What is the Scalability of Mingle?

In this section, we examine whether Mingle is able to scale with an increasing number of hosts. We consider scenarios with and without our security mechanism. We run the Mingle server on every host in a cluster of up to 23 computers. Each server has a precomputed index table of 100 MB text.

The figure below plots the average search latency and the standard deviations without security checking by varying the number of participating hosts. We can see that the performance degradation is not constant with the increasing number of Mingle servers. The search latency increases most when the number of hosts in Mingle increases from one to three. The increased latency is due to network communication and remote processing, which do not happen in the single server case. When we further increase the number of participating hosts, the performance degradation becomes smaller. The reason is that although the network communication time is increased, the remote processing can be done in parallel on different servers. We note that when the number of hosts is greater than 21, the search latency with security checking is even lower than that without security checking. This is because the security overhead is small compared with the overall search latency. The lower search latency with security checking is due to the large variance of network latencies when there are more hosts in the cluster. Overall, our measurements suggest that Mingle is able to scale with increasing number of hosts.



2.6 Discussion

We have developed Mingle to help authorized users efficiently locate their data in distributed computing environments. Mingle hosts precompute an inverted index of local files, searching among each other in a peer-to-peer way. The Mingle security architecture consists of authorization and authentication mechanisms. One of the major benefits of our security mechanism is user convenience. For authorization, we introduce an access-right mapping that allows data owners to conveniently specify access permissions. This is supported using a user authentication mechanism that permits a form of single sign-on.

Future work includes expanding Mingle to larger networks via hierarchically organizing clusters; considering schemes for encrypting and replicating host indexes; and better understanding Mingle's vulnerability to attacks such as timing attacks.

3 Byzantine-tolerant erasure-coded storage

Survivable storage systems spread data redundantly across a set of distributed storage-nodes in an effort to ensure its availability despite the failure or compromise of storage-nodes. Such systems require some protocol to maintain data consistency and liveness in the presence of failures and concurrency.

This section presents a new consistency protocol that operates in an asynchronous environment and tolerates Byzantine failures of clients and storage-nodes. The protocol supports a hybrid failure model in which up to t storage-nodes may fail: $b \leq t$ of these failures can be Byzantine and the remainder can be crash. The protocol requires at least $2t+2b+1$ storage-nodes. The protocol also supports use of m-of-n erasure codes (i.e., m-of-n fragments are needed to reconstruct the data), which usually require less network bandwidth (and storage space) than full replication [55][57].

Briefly, the protocol works as follows. To perform a write, a client determines the current logical time and then writes time-stamped fragments to at least a threshold quorum of storage-nodes. Storage-nodes keep all versions of fragments they are sent until garbage collection frees them. To perform a read, a client fetches the latest fragment versions from a threshold quorum of storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional and historical fragments are fetched, and repair may be performed, until a completed write is observed.

The protocol gains efficiency from five features. First, the space-efficiency of m-of-n erasure codes can be substantial, reducing communication overheads significantly. Second, most read operations complete in a single round trip: reads that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work. Most studies of distributed storage systems (e.g., [3][38]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in well under 1% of operations). Failures, although tolerated, ought to be rare. Third, incomplete writes are replaced by subsequent writes or reads (that perform repair), thus preventing future reads from incurring any additional cost; when subsequent writes do the fixing, additional overheads are never incurred. Fourth, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [22]. Fifth, the protocol only requires the use of cryptographic hashes, rather than more expensive cryptographic primitives (e.g., digital signatures).

This protocol is timely because many research storage systems are investigating practical means of achieving high fault tolerance and scalability. Examples include the FARSITE project at Microsoft Research [1], the Federated Array of Bricks project at HP Labs [16], and the OceanStore project at Berkeley [28]. Some of these projects (e.g., [1][28]) use Castro's Byzantine Fault Tolerant (BFT) library [7]. Many of these projects (e.g., [16][28]) are considering the use of

erasure codes for data storage. Our protocol for Byzantine-tolerant erasure-coded storage can provide an efficient, scalable, highly fault-tolerant foundation for such storage systems.

3.1 Background

In a decentralized storage system, multiple storage-nodes work together to implement a service for read-write storage. To write a data-item *D*, Client *A* issues write requests to multiple storage-nodes. To read *D*, Client *B* issues read requests to an overlapping subset of storage-nodes. This scheme provides access to data-items even when subsets of the storage-nodes have failed. One difficulty created by this architecture is the need for a consistent view, across storage-nodes, of the most recent update. Without such consistency, data loss is possible or even likely.

A common data distribution scheme used in such systems is replication, in which a writer stores a replica of the new data-item value at each storage-node to which it sends a write request. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. Alternately, more space-efficient erasure coding schemes can be used to reduce network load and storage consumption. With erasure coding schemes, reads require fragments from multiple servers. Moreover, to decode the data-item, the set of fragments read must correspond to the same write operation.

To provide reasonable semantics, storage systems must guarantee that readers see consistent data-item values. Specifically, the linearizability of operations is desirable for a shared storage system. Our protocol tolerates Byzantine faults of any number of clients and a limited number of storage nodes while implementing linearizable [21] and wait-free [19] read-write objects. Linearizability is adapted appropriately for Byzantine clients,[7] and wait-freedom is as in the model of Jayanti et al. [23].

As discussed in Section 1, most prior systems implementing Byzantine fault-tolerant services adopt the replicated state machine approach [48], whereby all operations are processed by server replicas in the same order (*atomic broadcast*). While this approach supports a linearizable, Byzantine fault-tolerant implementation of *any* deterministic object, such an approach cannot be wait-free [19][23]. Instead, such systems achieve liveness only under stronger timing assumptions, such as synchrony (e.g., [11][42][50]) or partial synchrony [12] (e.g., [7][24][45]), or probabilistically (e.g., [5]). An alternative is Byzantine quorum systems [31], from which our protocol inherit techniques (i.e., our protocol can be considered a Byzantine quorum system that uses the threshold

[7] Specifically, return values of reads by Byzantine clients are ignored, as are begin times of writes by Byzantine clients.

quorum construction). Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [8]), but also necessarily forsake wait-freedom to do so. Additionally, most protocols using Byzantine quorum systems utilize digital signatures, which are computationally expensive.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [20][32][35][41]. Of these related quorum systems, only Martin et al. [35] achieve linearizability in our fault model, and this work is also closest to ours in that it uses a type of versioning. In our protocol, a reader may retrieve fragments for several versions of the data-item in the course of identifying the return value of a read. Similarly, readers in [35] "listen" for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs by clients reading past versions, versus listening for future versions broadcast by servers. In our fault model, especially in consideration of faulty clients, our protocol has several advantages. First, our protocol works for erasure-coded data, whereas extending [35] to erasure coded data appears nontrivial. Second, ours provides better message efficiency: [35] involves a $\Theta(N^2)$ message exchange among the N servers per write (versus no server-to-server exchange in our case) over and above otherwise comparable (and linear in N) message costs. Third, ours requires less computation, in that [35] requires digital signatures by clients, which in practice is two orders of magnitude more costly than the cryptographic transforms we employ. Advantages of [35] are that it tolerates a higher fraction of faulty servers than our protocol, and does not require servers to store a potentially unbounded number of data-item versions. Our prior analysis of versioning storage, however, suggests that the latter is a non-issue in practice [52], and even under attack this can be managed using a garbage collection mechanism we discuss in Section 3.5.3.

There exists much prior work (e.g., [2][20][36]) that combines erasure coded data (e.g., [43][49]) with quorum systems to improve the confidentiality and/or integrity of data along with its availability. However, these systems do not provide consistency (i.e., an external synchronization mechanism is required) and do not cope with Byzantine clients.

We develop our protocol for a hybrid failure model of storage-nodes (i.e., a mix of crash and Byzantine failures). The concept of hybrid failure models was introduced by Thambidurai and Park [53]. Other protocols have been developed for such failure models: e.g., Garay and Perry [17] consider reliable broadcast, consensus and clock synchronization in the hybrid failure model and Malkhi, Reiter and Wool [33] consider the resilience of Byzantine quorum systems to crash faults.

3.2 System Model

We describe the system infrastructure in terms of *clients* and *storage-nodes*. There are *N* storage-nodes and an arbitrary number of clients in the system.

A client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution. A client or storage-node that deviates from its specification *fails*. We assume a hybrid failure model for storage-nodes. Up to *t* storage-nodes may fail, $b \leq t$ of which may be Byzantine faults [29]; the remainder are assumed to crash. We make no assumptions about the behavior of Byzantine storage-nodes and Byzantine clients (e.g., we assume that Byzantine storage-nodes can collude with each other and with any Byzantine clients). A client or storage-node that does not exhibit a Byzantine failure (it is either correct or crashes) is *benign*.

The protocol tolerates crash and Byzantine clients. As in any practical storage system, an authorized Byzantine client can write arbitrary values to storage, which affects the value of the data, but not its consistency. We assume that Byzantine clients and storage-nodes are computationally bounded so that we can benefit from cryptographic primitives.

We assume an asynchronous model of time (i.e., we make no assumptions about message transmission delays or the execution rates of clients and storage-nodes, except that it is non-zero). We assume that communication between a client and a storage-node is point-to-point, reliable, and authenticated: a correct storage-node (client) receives a message from a correct client (storage-node) if and only if that client (storage-node) sent it to it.

There are two types of *operations* in the protocol — *read operations* and *write operations* — both of which operate on *data-items*. Clients perform read/write operations that issue multiple read/write *requests* to storage-nodes. A read/write request operates on a *data-fragment*. A data-item is *encoded* into data-fragments. Clients may encode data-items in an erasure-tolerant manner; thus the distinction between data-item and data-fragment. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request *hosts* that write operation.

Storage-nodes provide fine-grained versioning; correct storage-nodes host a version of the data-fragment for each write request they execute. There is a well known zero time, 0, and null value, $\perp$, which storage-nodes can return in response to read requests. Implicitly, all stored data is initialized to $\perp$ at time 0.

3.3 Protocol

This section describes our Byzantine fault-tolerant consistency protocol that efficiently supports erasure-coded data-items by taking advantage of versioning storage-nodes. It presents the mechanisms employed to encode and decode data, and to protect data integrity from Byzantine storage-nodes and clients. It then describes the protocol in pseudo-code form. Finally, it develops constraints on protocol parameters that provide safety and liveness of the protocol.

3.3.1 Overview

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify data-fragments pertaining to the same write operation across the set of storage-nodes. For each write, a logical timestamp is constructed by the client that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). This is accomplished by querying storage-nodes for the greatest timestamp they host, and then incrementing the greatest response. In order to verify the integrity of the data, a hash that can verify data-fragment correctness is appended to the logical timestamp.

To perform a read operation, clients issue read requests to a subset of storage-nodes. Once at least a read quorum of storage-nodes reply, the client identifies the *candidate*—the response with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate comprise the *candidate set*. The read operation *classifies* the candidate as *complete*, *repairable*, or *incomplete*. If the candidate is classified as complete, the data-fragments, timestamp, and return value are validated. If validation is successful, the value of the candidate is returned and the read operation is complete; otherwise, the candidate is reclassified as incomplete. If the candidate is classified as repairable, it is repaired by writing data-fragments back to the original set of storage-nodes (note, in [32], repair, for replicas, is referred to as “write-back”). Prior to performing repair, data-fragments are validated in the same manner as for a complete candidate. If the candidate is classified as incomplete, the candidate is discarded, previous data-fragment versions are requested, and classification begins anew. All candidates fall into one of the three classifications, even those corresponding to concurrent or failed write operations.

3.3.2 Mechanisms

Several mechanisms are used in our protocol to achieve linearizability in the presence of Byzantine faults.

3.3.2.1 Erasure codes

In an erasure coding scheme, *N* data-fragments are generated during a write (one for each storage-node), and any *m* of those data-fragments can be used to decode the original data-item. Any *m* of the data-fragments can deterministically generate the other *N*–*m* data-fragments. We use a systematic information dispersal algorithm [43], which stripes the data-item across the first *m* data-fragments and generates erasure-coded data-fragments for the remainder. Other threshold erasure codes (e.g., secret sharing [49] and short secret sharing [27]) work as well.

3.3.2.2 Data-fragment integrity

Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to *b* storage-node integrity faults.

3.3.2.2.1 Cross checksums

Cross checksums [18] are used to detect corrupt data-fragments. A cryptographic hash of each data-fragment is computed. The set of *N* hashes are concatenated to form the *cross checksum* of the data-item. The cross checksum is stored with each data-fragment (i.e., it is replicated *N* times). Cross checksums enable read operations to detect data-fragments that have been modified by storage-nodes.

3.3.2.2.2 Write operation integrity

Mechanisms are required to prevent Byzantine clients from performing a write operation that lacks integrity. If a Byzantine client generates random data-fragments (rather than erasure coding a data-item correctly), then each of the *N*-choose-*m* permutations of data-fragments could "recover" a distinct data-item. Additionally, a Byzantine client could partition the set of *N* data-fragments into subsets that each decode to a distinct data-item. These attacks are similar to *poisonous writes* for replication as described by Martin et al. [35]. To protect against Byzantine clients, the protocol must ensure that read operations only return values that are written correctly (i.e., that are *single-valued*). To achieve this, the cross checksum mechanism is extended in three ways: validating timestamps, storage-node verification, and validated cross checksums.

3.3.2.2.3 Validating timestamps

To ensure that only a single set of data-fragments can be written at any logical time, the hash of the cross checksum is placed in the low order bits of the logical timestamp. Note, the hash is used for space-efficiency; instead, the entire cross checksum could be placed in the low bits of the timestamp.

3.3.2.2.4 Storage-node verification

On a write, each storage-node verifies its data-fragment against its hash in the cross checksum. The storage-node also verifies the cross checksum against the hash in the timestamp. A correct storage-node only executes write requests for which both checks pass. Thus, a Byzantine client cannot make a correct storage-node appear Byzantine. It follows, that only Byzantine storage-nodes can return data-fragments that do not verify against the cross checksum.

3.3.2.2.5 Validated cross checksums

Storage-node verification combined with a validating timestamp ensures that the data-fragments considered by a read operation were written by the client (as opposed to being fabricated by Byzantine storage-nodes). To ensure that the client that performed the write operation acted correctly, the reader must validate the cross checksum. To validate the cross checksum, all *N* data-fragments are required. Given any *m* data-fragments, the full set of *N* data-fragments a correct client should have written can be generated. The "correct" cross checksum can then be computed from the regenerated set of data-fragments. If the generated cross checksum does not match the verified cross checksum, then a Byzantine client performed the write operation. Only a single-valued write operation can generate a cross checksum that verifies against the validating timestamp. Instead of using validated cross checksums, our protocol could use verifiable secret sharing [9][13]. Verifiable secret sharing enables storage-nodes to validate that the client acted correctly on each write request (instead of validating the data-item on each read operation).

3.3.3 Pseudocode

The pseudocode for the protocol is shown below. The symbol LT denotes logical time and LTC denotes the logical time of the candidate. D_1, …,D_N denotes the *N* data-fragments; likewise, S_1, …,S_N denotes the set of *N* storage-nodes. In the pseudocode, the binary operator "|" denotes string concatenation. Simplicity and clarity in the presentation of the pseudo-code were chosen over obvious optimizations that could be used in an actual implementation.

```
1.  WRITE (Data):
2.    TIME := READ_TIMESTAMP()
3.    D1, …,DN := ENCODE(Data)
4.    CC :=
          MAKE_CROSS_CHECKSUM(D1,…,DN)
5.    LT := MAKE_TIMESTAMP(TIME, CC)
6.    DO_WRITE(D1, …,DN, LT, CC)

7.  READ_TIMESTAMP():
8.    forall Si ∈ { S1, …,SN }
9.      SEND(Si, TIME_REQUEST)
10.   ResponseSet := ∅
11.   repeat
12.     ResponseSet :=
              ResponseSet ∪
              { RCV(StorageNode,
                    TIME_RESPONSE) }
13.   until (|ResponseSet| = N − t)
14.   TIME :=
          MAX[ResponseSet.LT.TIME]
15.  RETURN (TIME)

16. MAKE_CROSS_CHECKSUM(D1, …,DN):
17.   forall Di ∈ D1, …,DN
18.     Hi := HASH(Di)
19.   CC := H1|…|HN
20.   RETURN(CC)

21. MAKE_TIMESTAMP(LTM, CC):
22.   LT.TIME := LTM.TIME + 1
23.   LT.Verifier := HASH(CC)
24.   RETURN(LT)

25. DO_WRITE(D1, …,DN, LT, CC):
26.   forall Si ∈ { S1, …,SN }
27.     SEND(Si, WRITE_REQUEST,
            LT, Di, CC)
28.   ResponseSet := ∅
29.   repeat
         ResponseSet :=
           ResponseSet ∪
             { RCV(StorageNode,
                   WRITE_RESPONSE) }
30.   until (|ResponseSet| = N − t)
```

Write operation pseudocode

```
1.  READ():
2.    ResponseSet :=
          DO_READ(READ_LATEST_REQUEST, ⊥)
3.    loop
4.      ⟨CandidateSet, LTC⟩ :=
          CHOOSE_CANDIDATE(ResponseSet)
5.      if (|CandidateSet| ≥ INCOMPLETE)
6.        /* Complete or repairable
             write found */
7.        D1, …,DN :=
            GENERATE_FRAGS(CandidateSet)
8.        VCC :=
            MAKE_CROSS_CHECKSUM(D1,…,DN)
9.        if (VCC = CandidateSet.CC)
10.         /* Cross checksum is
               validated */
11.         if (|CandidateSet|<COMPLETE)
12.           /* Repair is necessary */
13.           DO_WRITE(D1,…,DN,LTC,VCC)

14.         Data := DECODE(D1, …,DN)
15.         RETURN ( ⟨LTC, Data⟩ )

16.     /* Incomplete or cross checksum
           not validated, loop again */
17.     ResponseSet :=
          DO_READ(READ_PREV_REQUEST, LTC)

18. DO_READ(READ_COMMAND, LT):
19.   forall Si ∈ { S1, …,SN }
20.     SEND(Si, READ_COMMAND, LT)
21.   ResponseSet := ∅
22.   repeat
23.     Resp := RCV(S, READ_RESPONSE)
24.     if (VALIDATE(Resp.D, Resp.CC,
                     Resp.LT, S) = TRUE)
25.       ResponseSet :=
              ResponseSet ∪ { Resp }
26.   until (|ResponseSet| = N − t)
27.   RETURN(ResponseSet)

28. VALIDATE (D, CC, LT, S):
29.   if ((HASH(CC) ≠ LT.Verifier) OR
          (HASH (D) ≠ CC[S]))
30.     RETURN (FALSE)
31.   RETURN (TRUE)
```

Read operation pseudo-code

3.3.3.1 Storage-node interface

Storage-nodes offer interfaces to write a data-fragment at a specific logical time; to query the greatest logical time of a hosted data-fragment; to read the hosted data-fragment with the greatest logical time; and to read the hosted data-fragment with the greatest logical time before some given logical time. Given the

simplicity of the storage-node interface, storage-node pseudo-code has been omitted.

3.3.3.2 Write operation

The WRITE operation consists of determining the greatest logical timestamp, constructing write requests, and issuing the requests to the storage-nodes. First, a timestamp greater than, or equal to, that of the latest complete write must be determined. Collecting $N–t$ responses, on line 13 of READ_TIMESTAMP, ensures that the response set intersects a complete write at a correct storage-node. Since the environment is asynchronous, a client can wait for no more than $N–t$ responses. Fewer than $N–t$ responses are actually required to observe the timestamp of the latest complete write, since a single correct response is sufficient.

Next, the ENCODE function, on line 3 of WRITE, encodes the data-item into N data-fragments. The data-fragments are used to construct a cross checksum from the concatenation of the hash of each data-fragment (line 19). The function MAKE_TIMESTAMP, called on line 5, generates a logical timestamp to be used for the current write operation. This is done by incrementing the high order bits of the greatest observed logical timestamp from the ResponseSet (i.e., LT.TIME) and appending the Verifier. The Verifier is just the hash of the cross checksum.

Finally, write requests are issued to all storage-nodes. Each storage-node is sent a specific data-fragment, the logical timestamp, and the cross checksum. A storage-node validates the cross checksum with the verifier and validates the data-fragment with the cross checksum before executing a write request (i.e., storage-nodes call VALIDATE listed in the read operation pseudo-code). The write operation returns to the issuing client once $N–t$ WRITE_RESPONSE messages are received (line 30 of DO_WRITE).

3.3.3.3 Read operation

The read operation iteratively identifies and classifies candidates, until a repairable or complete candidate is found. Once a repairable or complete candidate is found, the read operation validates its correctness and returns the data. Note that the read operation returns a ⟨timestamp, value⟩ pair; in practice, a client only makes use of the value returned.

The read operation begins by issuing READ_LATEST_REQUEST commands to all storage-nodes (via the DO_READ function). Each storage-node responds with the data-fragment, logical timestamp, and cross checksum corresponding to the greatest timestamp it has executed.

The integrity of each response is individually validated through the VALIDATE function, called on line 24 of DO_READ. This function checks the cross

checksum against the Verifier found in the logical timestamp and the data-fragment against the appropriate hash in the cross checksum.

Since, in an asynchronous system, slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N-t$ read responses can be collected (line 26 of DO_READ). Since correct storage-nodes perform the same validation before executing write requests, the only responses that can fail the client's validation are those from Byzantine storage-nodes. For every discarded Byzantine storage-node response, an additional response can be awaited.

After sufficient responses have been received, a candidate for classification is chosen. The function CHOOSE_CANDIDATE, called on line 4 of READ, determines the candidate timestamp, denoted LTC, which is the greatest timestamp found in the response set. All data-fragments that share LTC are identified and returned as the candidate set. At this point, the candidate set contains a set of validated data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as complete, repairable, or incomplete based on the size of the CandidateSet. The rules for classifying a candidate as INCOMPLETE or COMPLETE are given in the following subsection. If the candidate is classified as incomplete, a READ_PREV_REQUEST message is sent to each storage-node with its timestamp. Candidate classification begins again with the new response set.

If the candidate is classified as either complete or repairable, the candidate set contains sufficient data-fragments written by the client to decode the original data-item. To validate the observed write's integrity, the candidate set is used to generate a new set of data-fragments (line 7 of READ). A validated cross checksum, VCC, is computed from the newly generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 9 of READ). If the check fails, the candidate was written by a Byzantine client; the candidate is reclassified as incomplete and the read operation continues. If the check succeeds, the candidate was written by a correct client and the read enters its final phase. Note that this check either succeeds or fails for all correct clients regardless of which storage-nodes are represented within the candidate set.

If necessary, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 13 of READ). Storage-nodes not currently hosting the write execute the write at the given logical time; those already hosting the write are safe to ignore it. Finally, the function DECODE, on line 14 of READ, decodes m data-fragments, returning the data-item.

It should be noted that, even after a write completes, it may be classified as repairable by a subsequent read, but it will never be classified as incomplete. For example, this could occur if the read set (of $N–t$ storage-nodes) does not fully encompass the write set (of $N–t$ storage-nodes).

3.4 Protocol constraints

The symbol Q denotes the number of benign storage-nodes that must execute write responses for a write operation to be complete. Note that since threshold quorums are used, Q is a scalar value. To ensure that linearizability and liveness are achieved, Q and N must be constrained with regard to b, t, and each other. As well, the parameter m, used in DECODE, must be constrained.

3.4.1 Write termination

To ensure write operations are able to complete in an asynchronous environment,

$$Q \leq N - t - b$$

Since slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N - t$ responses can be awaited. As well, b responses received may be from Byzantine storage-nodes.

3.4.2 Read classification

To classify a candidate as COMPLETE, a candidate set of at least Q benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|\text{CandidateSet}| - b \geq Q \Rightarrow \text{COMPLETE}$$

To classify a candidate as INCOMPLETE a client must determine that a complete write does not exist in the system (i.e., fewer than Q benign storage-nodes host the write). For this to be the case, the client must have queried all possible storage-nodes ($N - t$), and must assume that nodes not queried host the candidate in consideration. So,

$$|\text{CandidateSet}| + t < Q \Rightarrow \text{INCOMPLETE}$$

3.4.3 Real repairable candidates

To ensure that Byzantine storage-nodes cannot fabricate a repairable candidate, a candidate set of size b must be classifiable as incomplete. Substituting b into

$$|\text{CandidateSet}| + t < Q \Rightarrow \text{INCOMPLETE}$$

yields

$$b + t < Q$$

3.4.4 Decodable repairable candidates

Any repairable candidate must be decodable. The lower bound on candidate sets that are repairable follows from the relations above, since the upper bound on classifying a candidate as incomplete coincides with the lower bound for classifying it as repairable:

$$1 \leq m \leq Q - t$$

3.4.5 Constraint summary

The constraints developed the previous section can be summarized as follows:

$$|\text{CandidateSet}| \geq Q + b \Rightarrow \text{COMPLETE}$$
$$|\text{CandidateSet}| < Q - t \Rightarrow \text{INCOMPLETE}$$
$$t + b + 1 \leq Q \leq N - t - b$$
$$2t + 2b + 1 \leq N$$
$$1 \leq m \leq Q - t$$

3.5 Discussion

3.5.1 Byzantine clients

In a storage system, Byzantine clients can write arbitrary values. The use of fine-grained versioning (e.g., self-securing storage [52]) facilitates detection, recovery, and diagnosis from storage intrusions. Once discovered, arbitrarily modified data can be rolled back to its pre-corruption state.

Byzantine clients can also attempt to exhaust the resources available to the protocol. Issuing an inordinate number of write operations could exhaust storage space. However, continuous garbage collection frees storage space “behind” the latest complete write. (See Section 3.5.3.) If a Byzantine client were to intentionally issue incomplete write operations, then garbage collection may not be able to free up space. In addition, incomplete writes require read operations to roll-back behind them, thus consuming client computation and network resources. In practice, storage-based intrusion detection [39] is probably sufficient to detect such client actions.

3.5.2 Timestamps from Byzantine storage-nodes

Byzantine storage-nodes can fabricate high timestamps that must be classified as incomplete by read operations. Worse, in each subsequent round of a read operation, Byzantine storage-nodes can fabricate more high timestamps that are just a bit smaller than the previous. In this manner, Byzantine storage-nodes can "attack" the performance of the read operation, but not its safety. To protect against such denial-of-service attacks, the read operation can consider all unique timestamps, up to a maximum of b+1, present in a ResponseSet as candidates before soliciting another ResponseSet. In this manner, each "round" of the read operation is guaranteed to consider at least one candidate from a correct storage-node and no more than b candidates from Byzantine storage-nodes.

3.5.3 Garbage Collection

The wait-freedom of our protocol utilizing versioning relies on data versions not exceeding the available storage capacity. If storage capacity is exhausted, wait-freedom cannot be guaranteed. Prior experience indicates that it takes weeks of normal activity to exhaust the capacity of modern disk systems that version all write requests [52].

Garbage collection is used to avoid storage exhaustion. Briefly, garbage collection works by each storage-node periodically performing a READ-like protocol to determine the latest complete write per data-item, and then freeing local versions that precede the latest complete write. In doing so, however, it can interact with concurrent read operations and concurrent write operations in such a manner that a read operation must be retried. Specifically a read operation could classify a concurrent write operation as incomplete, the write operation could then complete, and garbage collection could then delete all previous complete writes. If this occurs, the read operation's next round will observe an incomplete write with no previous history. Effectively, the read operation has "missed" the complete write operation that it would have classified as such. When it discovers this fact, the read operation retries (i.e., restarts by requesting a new ResponseSet). Thus, in theory, a read operation faced with perpetual write concurrency and garbage collection may never complete. In practice, such prolonged interaction of garbage collection and read-write concurrency for a given data-item should occur rarely, if ever.

4 Bibliography

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2002.

[2] D. Agrawal and A. El Abbadi. Integrating security with fault-tolerant distributed databases. *Computer Journal* 33(1):71-78, 1990.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the ACM Symposium on Operating System Principles*, October 1991.

[4] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. Harvest: A scalable, customizable, discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, 1994.

[5] C. Cachin, K. Kursawe, F. Petzold and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology – CRYPTO 2001*, pages 524-541, 2001.

[6] J. Callan. Distributed information retrieval. In W. B. Croft, ed., *Advances in Information Retrieval*. Kluwer Academic Publishers, pages 127-150, 2000.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems* 20(4):398-461, November 2002.

[8] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 11–20, April 2001.

[9] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing in the presence of faults. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 335-344, 1985.

[10] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability* (Lecture Notes in Computer Science 2000), December 2000.

[11] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. *Information and Computation* 118(1):158-179, April 1995.

[12] C. Dwork, N. Lynch and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM* 35(2):288-323, April 1988.

[13] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 427-437, 1987.

[14] W. Ford and B. Kaliski. Server-assisted generation of a strong secret from a password. In *Proceedings of the 9th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2000.

[15] J. French, A. Powell, J. Callan, C. Viles, T. Emmitt, K. Prey, and Y. Mou. Comparing the performance of database selection algorithms. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 238-245, 1999.

[16] S. Frolund, A. Merchant, Y. Saito, S. Spence and A. Veitch. FAB: Enterprise storage systems on a shoestring. In *Proceedings of Hot Topics in Operating Systems*, May 2003.

[17] J. A. Garay and K. J. Perry. A continuum of failure models for distributed computing. In *Proceedings of the International Workshop on Distributed Algorithms*, pages 153-165, 1992.

[18] L. Gong. Securely replicating authentication services. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 85-91, 1989.

[19] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages* 13(1):124-149, 1991.

[20] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In *Advances in Cryptology – CRYPTO 1987*, pages 279-391, 1987.

[21] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463-492, July 1990.

[22] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham and M. J. West. Scale and

performance in a distributed file system. *ACM Transactions on Computer Systems* 6(1):51-81, February 1988.

[23] P. Jayanti, T. D. Chandra and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM* 45(3):451-500, May 1998.

[24] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security* 1(4):371-406, November 2001.

[25] S. T. Kirsch. Document retrieval over networks wherein ranking and relevance scores are computed at the client for multiple database documents. U. S. Patent 5,659,732, 1997.

[26] D. Klein. Foiling the cracker: A survey of, and improvement to, password security. In *Proceedings of the 2nd USENIX Security Workshop*, August 1990.

[27] H. Krawczyk. Secret sharing made short. In *Advances in Cryptology – CRYPTO 93*, pages 136-146, 1994.

[28] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaten, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[29] L. Lamport, R. Shostak and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382-401, July 1982.

[30] B. Lampson, M. Abadi, M. Burrows and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10(4):265-310, November 1992.

[31] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing* 11(4):203–213, 1998.

[32] D. Malkhi and M. K. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, October 1998.

[33] D. Malkhi, M. Reiter and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing* 29(6):1889-1906, April 2000.

[34] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report 34, Department of Computer Science, The University of Arizona, 1993.

[35] J.-P. Martin, L. Alvisi and M. Dahlin. Minimal byzantine storage. In *Proceedings of the International Symposium on Distributed Computing*, October 2002.

[36] R. Mukkamala. Storage efficient and secure replicated distributed databases. *IEEE Transactions on Knowledge and Data Engineering* 6(2):337-341, 1994.

[37] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM* 22(11):594-597, November 1979.

[38] B. D. Noble and M. Satyanarayanan. An emperical study of a highly available file system. Technical Report CMU-CS-94-120, School of Computer Science, Carnegie Mellon University, February 1994.

[39] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson and G. R. Ganger. Storage-based intrusion detection: watching storage activity for suspicious behavior. In *Proceedings of the USENIX Security Symposium*, August 2003.

[40] R. Perlman and C. Kaufman. Secure password-based protocol for downloading a private key. In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, February 1999.

[41] E. T. Pierce. Self-adjusting quorum systems for byzantine fault tolerance. Technical Report CS-TR-01-07, Department of Computer Science, University of Texas at Austin, March 2001.

[42] F. M. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems* 7(1):25-60, February 1989.

[43] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM* 36(2):335-348, April 1989.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, August 2001.

[45] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[46] R. L. Rivest, A. Shamir and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 27(2), February 1978.

[47] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329-350, 2001.

[48] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys* 22(4):299-319, December 1990.

[49] A. Shamir. How to share a secret. *Communications of the ACM* 22(11):612-613, November 1979.

[50] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs and A.Tully. Principal features of the voltan family of reliable node architectures for distributed systems. *IEEE Transactions on Computers* 41(5):542-549, May 1992.

[51] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, August 2001.

[52] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules and G. R. Ganger. Self-securing storage: protecting data in compromised systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, October 2000.

[53] P. Thambidurai and Y. Park. Interactive consistency with multiple failure modes. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 93-100, October 1988.

[54] C. L. Viles and J. C. French. Dissemination of collection wide information in a distributed information retrieval system. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 12-20, 1995.

[55] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[56] I. H. Witten, A. Moffet and T. C. Bell. *Managing gigabytes: Compressing and indexing documents and images*. 2nd Edition, Morgan Kaufmann, 1999.

[57] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote and P. K. Khosla. Survivable information storage systems. *IEEE Computer* 33(8):61-68, August 2000.

[58] Y. Zhao, J. D. Kubiatowicz and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, April 2000.